

## ITS assignment 2

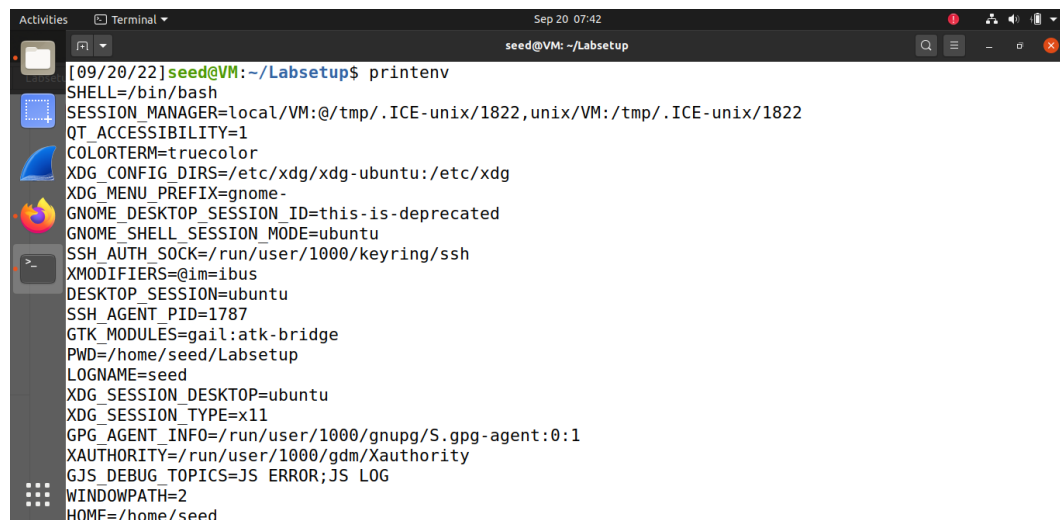
xdr622 and rsk975

September 2022

# SEED

## Task 1

In this task we had to get familiar with the `printenv` and `export` commands. We tested this by unsetting the `COLORTERM` environment variable. The `printenv` was used to show all the environment variables while `export` or `unset` was used for unsetting or changing the variable. Environment variables affect running processes and store information about user settings and paths to temporary files.

A terminal window titled "Terminal" with a date and time of "Sep 20 07:42". The prompt is "seed@VM: ~/Labsetup". The command "printenv" has been executed, displaying a list of environment variables. The output is as follows:

```
[09/20/22]seed@VM:~/Labsetup$ printenv
SHELL=/bin/bash
SESSION_MANAGER=local/VM:@/tmp/.ICE-unix/1822,unix/VM:/tmp/.ICE-unix/1822
QT_ACCESSIBILITY=1
COLORTERM=truecolor
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
XDG_MENU_PREFIX=gnome-
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
GNOME_SHELL_SESSION_MODE=ubuntu
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
XMODIFIERS=@im=ibus
DESKTOP_SESSION=ubuntu
SSH_AGENT_PID=1787
GTK_MODULES=gail:atk-bridge
PWD=/home/seed/Labsetup
LOGNAME=seed
XDG_SESSION_DESKTOP=ubuntu
XDG_SESSION_TYPE=x11
GPG_AGENT_INFO=/run/user/1000/gnupg/S.gpg-agent:0:1
XAUTHORITY=/run/user/1000/gdm/Xauthority
GJS_DEBUG_TOPICS=JS ERROR;JS LOG
WINDOWPATH=2
HOME=/home/seed
```

Figure 1: Excerpt of `printenv`

## Task 2

In this task we had to execute two C-programs, one using a child processor and one using the parent processors. We had to find out if the parent's environment variables are inherited by the child process. We first executed the program only with the parent process and stored the output. Afterwards we executed the program again with a child process and stored the output. We found using the `diff` command that the two outputs were identical. This means the child process does indeed inherit environment variables from the parent process.

## Task 3

In this task we have to test if a calling process inherits environment variables. We started by compiling and executing the given `myevn.c`. This did not do anything, which means no environment variables was printed in the terminal. We then rewrote `myevn.c` by changing `execve("/usr/bin/env", argv, NULL);` to `execve("/usr/bin/env", argv, environ);`. This change actually printed the current environment variables in the terminal.

The reason for the program not working when using `NULL` as the third argument is because the third argument is an `envp`. `envp` either has to be an array in the form `key=value` or the external `environ` variable. `NULL` is neither, thus the `execve()` function did not pass anything to the new process, which in resulted in no supplied environment variables.

The external variable `environ` is a variable that points to an array of pointers called the *environment*.

## Task 4

The `system()` command forks a child process that executes a new shell, then runs the command given as a string input. For example, our program supplies the input `"/usr/bin/env"` to `system()`; this command is equivalent to the shell command `exec1("/bin/sh", "sh", "-c", "/usr/bin/env", (char *) NULL)`.

In turn, `exec1()` calls `execve()` (similarly to Task 3), wherein a list of pointers to strings `"sh", "-c", "/usr/bin/env"` and a terminating `NULL` pointer are supplied as an argument to `execve()`'s `char *const argv[]` parameter.

From the manpage: "All other `exec()` functions (which do not include 'e' in the suffix) take the environment for the new process image from the external variable `environ` in the calling process." Thus, `exec1()` supplies the environment variable in precisely the same way as was manually done in Task 3.

## Task 5

In this task we have to learn about set-UID's and their processes. Firstly we compile the c-program `uid.c` that prints all the current process environment variables to the terminal. The file is compiled with the command: `gcc uid.c -o uid` The next step is to change the ownership and make it a set-UID program. This is done with the commands; `sudo chown root uid` and `sudo chmod 4755 uid`. `chown` changes the file owner and group. `chmod` is used to set the mode bits for the owner, the group of the owner and others. As seen in the figure the owner because root and the mode bits becomes `-rwsr-xr-x` where `r` means read, `w` means write, `x` means execute and `s` means setuid. We set the three environment variables: `PATH`, `LD_LIBRARY_PATH` and `HUTA0`. `HUTA0` is a self-made environment variable created using `export HUTA0="genshin"`. We started by setting the `PATH` using `export PATH=/usr/bin`. This made the UID program

```
| -rwsr-xr-x 1 root seed 16768 Sep 20 04:28 uid2.out
```

Figure 2: overview of modebits and owner

not run, and instead giving us the error message "*uid.out: command not found*". We then changed the PATH variable back to its original state and reexecuted the program. Now the environment variables were printed to the terminal including our home-made variable HUTAO="genshin", as seen in the figure.



```
seed@VM: ~/Labsetup
TERM=xterm-256color
LESSOPEN=| /usr/bin/lesspipe %s
USER=seed
GNOME_TERMINAL_SERVICE=:1.122
DISPLAY=:0
SHLVL=1
QT_IM_MODULE=ibus
HUTAO=genshin
XDG_RUNTIME_DIR=/run/user/1000
JOURNAL_STREAM=9:32616
XDG_DATA_DIRS=/usr/share/ubuntu:/usr/local/share:/usr/share:/var/lib/snapd/desktop
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:.
GDMSESSION=ubuntu
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus
OLDPWD=/home/seed
= ./uid2.out
109/20/221 seed@VM: ~/Labsetup$
```

Figure 3: home-made environment variable

## Task 6

First we created a new program called `ls`. The code simply echoes a string to the terminal. By changing the owner of the file as in Task 5, we give the file root permissions, and as a `Set-UID` program, these permissions are also granted to any user running the program.

In addition, we changed the path variable with `export PATH=/home/seed/LabSetup/src:$PATH` in order to point to the folder containing our code.

By executing `ls` in the command, the user executes our self-defined program instead of the typical `ls` command as defined by Linux in `/bin/dash`.

We made two versions of this `ls` command, one that ran `ls` but without a output, thus slowing the terminal. And another that used the `echo`-command to display text.

```

real ls [09/20/22] seed@VM:~/.../src$ ls
ls      ls.c  myenv.c  myprintenv.c  system.c  uid.c
echo [09/20/22] seed@VM:~/.../src$ ls
lol you been hacked bro, get over it
our ls [09/20/22] seed@VM:~/.../src$ ls
[09/20/22] seed@VM:~/.../src$ █

```

Figure 4: our malicious script in progress

## Short answers

### Question 1

*List the three main user authentication categories.*

The three main categories can be summed up as: what you know, what you have and what you are. What you know can be passwords, it is something that only you as a user knows and makes up yourself. It can also be password reset questions such as "the name of your first pet". What you have is biometric data such as your fingerprint or your iris. What you are is behavioral data, such as the way you talk or move your mouse.

### Question 2

*Briefly explain how passwords are stored securely.*

Conceptually, the relationship between usernames  $u_i$  and passwords  $p_i$  can be represented in a relation  $u_i \rightarrow p_i$ . These relations are usually stored as a tuple  $u_i, h_i$  where  $h_i = H(p_i)$  is a password hash; the output of the hash function with the password given must match the password hash stored in the file.

For even more security, password hashing can be combined with a salt. The previous userid-password table now consists of tuples  $(u_i, s_i, h_i)$ , where  $s_i$  is a salt – a randomly-selected  $t$ -bit value – and  $h_i = H(p_i, s_i)$ ; here, the password  $p_i$  and salt  $s_i$  are concatenated before being used as input to the hash function.

Peppering is similar to salting. For each password, a hash  $h_i = H(p_i, r_i)$  is created, then  $r_i$  is deleted. When matching a password with a password hash, the system tests all possible values of  $1 \leq r_i \leq R$  until a match is found. Peppering can be combined with salting as  $H(p_i, s_i, r_i)$  for additional attack slowdown.

### Question 3

*How does RBAC compare to DAC and MAC?*

RBAC requires defining the different roles in an organization and determining which privileges each role requires (and to what degree) in order to fulfill their necessary tasks. DAC is more decentralised than RBAC, as each user can individually determine which other users/groups may access files created by them. MAC bases access control on two elements: classifications and compartments. Users must have clearance for their corresponding ranks in order to access said files. This is in contrast to basing access control on roles (RBAC).

### Question 4

*XSS might allow an attacker to steal browser cookies, why?*

If the security on a site is not up to date, it is possible to trick the server into thinking something is Javascript and run it; the precise mechanism in which this achieved varies on the type of attack, but typically these attacks exploits server PHP code neglecting to sanitize input from the URL bar. When a user clicks the link (thinking it is something else like an image) a script will run in the background and steal the cookies.